Patent Application
Attorney Docket No.: 16159/023001; P6425

APPLICATION

FOR

UNITED STATES LETTERS PATENT

TITLE: **APPLICATION-INDEPENDENT API FOR DISTRIBUTED COMPONENT COLLABORATION**

APPLICANT: **Syed M. Ali, Robert N. Goldberg, Yury Kamen, Bruce K. Daniels, and Peter A. Yared**

"EXPRESS MAIL" Mailing Label Number: EV042548570US

Date of Deposit: December 3, 2001

APPLICATION-INDEPENDENT API FOR DISTRIBUTED COMPONENT COLLABORATION

Background of Invention

Field of the Invention

[0001] The invention relates generally to object-oriented technology. More specifically, the invention relates to collaboration between components in a distributed application.

Background Art

[0002] Modern enterprise applications are typically implemented as multi-tier systems. Multi-tier systems serve the end-user through a chain of client/server pairs. Enterprise systems are typically implemented in a number of components, where each component may contain multiple object instances at runtime. Each component interacts with other system components at runtime to provide a set of functions to the system.

[0003] Figure 1 shows an example of a four-tiered system that includes a user interface tier **2**, a web server tier **4**, an application server tier **6**, and a data tier **8**. The user interface tier **2** is the layer of interaction and typically includes a form-like graphical user interface (GUI) displayed by a display component, typically a web browser **10**. The web server tier **4** includes web components **12** hosted on a web server **14**. The web components **12** generate the content displayed by the web browser **10**. The application server tier **6** includes application components **16** hosted on an application server **18**. The application components **16** model the business rules, typically through interaction with application data. The data tier **8** includes a persistent data store, typically a database management system (DBMS) **20** and a database **22**.

[0004] The web browser **10** and the web server **14** form a client/server pair. The web server **14** and the application server **18** form another client/server pair. The application server **18** and DBMS **20** form yet another client/server pair. A web component **12** and an application component **16** are in a client/server relationship when the web component **12**

(client) uses services of the application component **16** (server) to provide functions to the system. In order for the client and server to collaborate, there must be a contract, or interface definition, between the client and server that specifies the server methods that can be invoked by the client. When the client and server are in different address spaces, the client uses some form of remote procedure call (RPC) to invoke the server methods. Typically, this involves the client calling into a local stub, which forwards the call to the server.

[0005] The granularity of an object is a measure of the size of the object and the number of the interactions the object makes with other objects. Large-grained objects have few interactions with other objects, while fine-grained objects have many interactions with other objects. Object-oriented, client/server programmers often develop server programs that embed large-grained and fine-grained object models. One of the consequences of fine-grained behavior is that the client program makes excessive remote method calls to the fine-grained object in the server program in order to access and update the attributes of the object. Remote method calls are expensive in terms of computer processor usage, input/output access, and overall time usage. For each remote method call, data may have to be marshaled and later un-marshaled, authentication may have to be performed before the client can use services provided by the server, packets may need to be routed through switches, and so forth. Thus, numerous remote method calls can have a huge impact on the performance and scalability of the application. For optimal distribution performance, the number of client/server roundtrips must be minimized.

[0006] The process of developing and deploying components in a distributed environment, such as that illustrated in Figure 1, is often iterative and incremental. It is not unusual to alter the different components of the distributed system a few times before the components reach a stable state. These modifications may include changing the interface and behavior of the objects in the components. Also, due to lack of distributed debuggers, it is common to inject code into the distributed application for the purpose of debugging the application.

[0007] Difficulties arise when modifications are made to the distributed application while the application is running. For example, if the interface or behavior of a remote (server) object is changed, all the parts of the distributed application have to be halted in order to re-deploy the modified remote object and its respective stubs on all the clients. Re-deployment resets the running state of the distributed application, requiring the application to be restarted. This makes deployment of distributed components extremely dependent on each other. Also, every time a new debug statement is injected into the distributed system, the entire application has to be re-deployed, making the edit, compile, and deploy cycle tedious.

[0008] At this point, there is no known work that allows distributed applications to run seamlessly after the interface between the components of the application has been modified. Most object request brokers (ORBs) and application servers support hot deployment, which allows modifications made to distributed applications to take effect without bringing the servers down. However, these deployment mechanisms typically reset the running state of the applications, resulting in the client stubs going "stale."

Summary of Invention

[0009] In one aspect, the invention relates to a method for a system having distributed collaborating components. The method comprises restricting direct interaction between distributed collaborating components by introducing an application-independent interface between distributed collaborating components and invoking a service from the application-independent interface in order to enable interaction between distributing collaborating components.

[0010] In one aspect, the invention relates to a method for a distributed system having a client and a server. The collaboration method comprises interposing a service layer between the client and the server. The service layer has a capability to interpret a specification from the client at runtime in order to enable interaction between the client and the server. The collaboration method further includes routing correspondence between the client and server through the service layer.

[0011] In one aspect, the invention relates to a computer-readable medium having recorded thereon instructions executable by a processor. The instructions are for receiving a specification from a client component and interpreting the specification in order to enable interaction between the client component and the server component.

[0012] In one aspect, the invention relates to a distributed system which comprises a client component, a server component having at least one object at runtime, and a service layer between the client and the server component. The service layer has a capability to interpret a specification of usage of the object at runtime.

[0013] In one aspect, the invention relates to a distributed system which comprises a service means for providing application-independent services and for interpreting a usage specification and a logic execution specification. The distributed system further includes a client component that sends the usage specification and the logic execution specification to the service means and a server component that interacts with the service means in order to provide services to the client component.

[0014] In one aspect, the invention relates to an apparatus for distributed collaborating components. The apparatus comprises a means for restricting direct interaction between distributed collaborating components by introducing an application-independent interface between distributed collaborating components, and means for invoking a service from the application-independent interface in order to enable interaction between distributing collaborating components.

[0015] Other features of the invention will be apparent from the following description and the appended claims.

Brief Description of Drawings

[0016] Figure 1 shows an example of a multi-tiered system.

[0017] Figure 2 shows a distributed system incorporating an embodiment of the invention.

[0018] Figure 3 shows the generic service layer of Figure 2 implemented as a session bean.

Detailed Description

[0019] A generic service layer consistent with the principles of the invention enables a distributed application to run seamlessly even after the interface between the components of the application has been modified. The generic service layer has a capability to interpret a usage specification and a logic execution specification at runtime. A usage specification specifies data (object attributes) required by a client, and a logic execution specification specifies services (object methods) required by a client. The invention separates usage specification and logic execution specification from implementation of the components and restricts all interactions between the components by introducing the generic service layer between them. At runtime, all correspondences between the client and server are routed through the generic service layer, which interprets the client's usage and logic execution specifications. Since all interactions are in the form of usage and logic execution specifications, all the round trips of fetching/posting data to the server can be done together in one round trip, instead of numerous round trips.

[0020] The following is a detailed description of specific embodiments of the invention. In the following detailed description, numerous specific details are set forth in order to provide a more thorough understanding of the invention. However, it will be apparent to one of ordinary skill in the art that the invention may be practiced without these specific details. In other instances, well-known features have not been described in detail to avoid obscuring the invention. For example, the various mechanisms for making calls to remote objects are not discussed. It will be clear to those skilled in the art that when objects run in separate address spaces some form of remote procedure call is needed to invoke the methods of the remote object.

[0021] Figure 2 shows a distributed system that includes a client **24** and a server **26**. The distributed system may include several other client/server pairs. However, it is not necessary to show several client/server pairs to illustrate the principles of the invention.

A client component **28** is deployed on the client **24**. The client component **28** may include one or more client objects **30**. A server component **32** is deployed on the server **26**. The server component **32** may include one or more server objects **34**. As an example, the client **24** may be a web server or other computer system, and the server **26** may be an application server or other computer system.

[0022] In accordance with an embodiment of the invention, a generic service layer **36** is interposed between the client component **28** and the server component **32** to prevent direct interaction between the components **28**, **32**. The service layer **36** is called "generic" because it provides application-independent services, *i.e.*, the generic service layer **36** provides services that can be reused by any application without modification. At runtime, the generic service layer **36** interprets usage and logic execution specifications, which are application-dependent, in order to enable collaboration between the client component **28** and the server component **32**. Examples of application-independent services provided by the generic service layer **36** include fetching data from an object, updating an object with given data, and executing logic.

[0023] In one embodiment, the generic service layer **36** fetches a list of objects and object attributes from the server component **32** based on a usage specification from the client component **28**. An example of a usage specification for an employee object *e* is shown below, where the terms "RW" and "R" stand for "Readable/Writeable" and "Readable," respectively.

```
e     "Employee Information" :  RW {
      empNo        "Employee Number": R,
      lastName     "Last Name",
      extension    "Extension",
      hireDate     "Hire Date",
      dob          "Date of Birth",
      manager      "Manager": R {
            empNo        "Manager Employee No",
      },
      department   "Department": R {
            name         "Department Name",
      },
      building     "Building": R {
            buildingName       "Building Name",
      }
}
```

[0024] In one embodiment, the generic service layer **36** interprets logic execution specification from the client component **28**. The logic execution specification includes a sequence of operations to be performed in the server component **32**. As an example, the logic execution specification may include instructions for creating a server object or for invoking certain business method calls on a server object. The following is an example of a logic execution specification for creating a new employee:

```
theManager = Managers.getSelectedOne();
theDepartment = Departments.getSelectedOne();
theBuilding = Buildings.getSelectedOne();

newEmployeePK = factory(NameTool.Employee).assignNextPK();
newEmployee = factory(NameTool.Employee).create(newEmployeePK);

newEmployee.setManger(theManager);
newEmployee.setDepartment(theDepartment);
newEmployee.setBuilding(theBuilding);

return new EmployeeDetail(newEmployee);
```

As shown in the usage specification above, the employee object *e* includes references to a manager object, department object, and building object. The logic execution specification includes instructions for locating selected manager, department, and building objects, creating a new Employee object and assigning a manager, a department, and a building to the newly created Employee object.

[0025] In operation, the client component **28** invokes services from the generic service layer **36** in order to interact with the server component **32**. To fetch data from a server object **34**, the client component **28** invokes a service from the generic service layer **36** that takes a usage specification as an argument. The usage specification would include the attributes of the server object **34** needed by the client component **28**. The generic service layer **36** finds the server object **34**, interprets the usage specification to determine the object attributes to fetch, fetches the requested attributes from the server object **34**, and returns the data to the client component **28**. The data is cached in a proxy **34S** for the server object **34** and locally accessed by the client component **28**. The client component **28** can modify the data cached in the proxy **34S** as needed.

[0026] To update the data in the server object **34** with the data in the proxy **34S**, the client component **28** invokes a service from the generic service layer **36** that takes a

usage specification and modified data in the proxy **34S** as arguments. The usage specification would include the attribute of the server object **34** to be updated with the modified data from the proxy **34S**. The generic service layer **36** finds the server object **34** again and updates the server object **34** using the modified data from the proxy **34S**.

[0027] If there is a need to execute any business logic, the client component **28** may also invoke a service of the generic service layer **36** that takes a logic execution specification as argument. The generic service layer **36** executes the logic and returns the results to the client component **28**. The logic execution specification can be modified as necessary at runtime. Thus, if the interface of a server object changes for any reason, the logic can be modified to reflect this change. The application can continue to run without having to reset its running state because the logic execution specification is interpreted at runtime by the generic service layer **36**. Debugging code can also be injected into the logic execution specification and the debugging code can be executed without having to first halt the application and restart the application.

[0028] The generic service layer **36** can be implemented in a variety of ways depending on the underlying technology of the distributed system. For example, in a Java™ 2 Platform, Enterprise Edition (J2EE™) architecture, the generic service layer **36** may be implemented as an Enterprise JavaBean™ (EJB™) component. For example, the generic service layer **36** may be implemented as a session bean. A session bean is created by the client and usually exists for the duration of the client/server session. The session bean performs operations on behalf of the client. A session bean can either be stateless or stateful, *i.e.*, maintain conversational state across methods and transactions. The client interacts with the bean through two wrapper interfaces: home interface and remote interface. The home interface is used for lifecycle operations, *i.e.*, creating, destroying, or finding instances of the session bean. The remote interface provides access to methods of the session bean. Complete information regarding the J2EE™ architecture and EJB is available from Sun Microsystems, both in print and via the Internet at java.sun.com.

[0029] Figure 3 shows the generic service layer **36** implemented as a session bean. The session bean **36** is hosted inside an EJB container **40** on an EJB server **42**. The EJB

server **42** is basically an application server that provides the environment necessary for execution of EJB applications. The server objects **34** are also hosted inside the EJB container **40**. This assumes that the server objects **34** are EJB components. If the server objects **34** are not EJB components, the session bean **36** may interact with the server objects **34** through an appropriate wrapper interface. The client side of the distributed system remains as described in Figure 2.

[0030] In one embodiment, the remote interface of the session bean provides access to the following services: getObjects, setObjects, invokeRemoteMethod, and makeTransition.

[0031] The getObjects method takes a server identity of the root of an object graph and a vector of usage strings defining what is to be fetched from the server as parameters. The getObjects method locates the object graph using the server identity and returns an object graph to the client that contains the attributes specified in the usage strings. There are various mechanisms for packaging object graphs for transport, such Java™ serialization and CORBA ("Common Object Request Broker Architecture").

[0032] The setObjects method takes an object graph to be stored on the server and a vector of usage strings defining what the client has changed as parameters. The setObjects method updates the server objects having the attributes specified in the vector of usage strings using the data stored in the object graph.

[0033] The invokeRemoteMethod method takes a remote method, a client-side object graph needed to execute the remote method, a vector of objects to update on the client after executing the remote method, and a vector of usage strings defining what to update on the client as parameters. The invokeRemoteMethod method updates the client-side object graph using the server data, invokes the remote method on the server, packages the result as well as the objects to update on the client, and sends the package to the client.

[0034] The makeTransition method stores an object graph and a vector of usage into a database. The vector of usage contains strings defining what has changed within the object graph. Further, the makeTransition method retrieves data using an array of usage information for all possible destination states that may be transitioned to. Finally, the

makeTransition method takes as a parameter business logic to be executed *e.g.,* Java™ script or a script written in another language, which is to be run on the server as part of the transition. The makeTransition method facilitates transition from one state to another by retrieving and storing object graphs based on the usage specification. In addition, the makeTransition method executes business logic on the server to determine the destination state of the transition using the array of usage information.

[0035] The invention provides one or more advantages. Separation of usage specification and logic execution specification makes development and debugging process of distributed components easier. Since all client/server interactions are in the form of usage and logic specifications, all the round trips of fetching/posting data to the server and logic execution can be done together in one trip, instead of multiple trips. Different distributed components can be developed without the knowledge of their cooperation with other components. Because the distributed components do not interact directly, it is possible to change any part of the distributed application without affecting the remaining system. Because of the interpreted nature of the execution of logic, it is possible to modify logic at runtime. Modification of logic execution at runtime also makes the debugging of distributed application easier. That is, debugging code can be injected into the logic execution specification and executed without having to re-deploy the application.

[0036] While the invention has been described with respect to a limited number of embodiments, those skilled in the art, having benefit of this disclosure, will appreciate that other embodiments can be devised which do not depart from the scope of the invention as disclosed herein. Accordingly, the scope of the invention should be limited only by the attached claims.